Universität Leipzig – Softwaretechnik Praktikum 2014 - xodx

Qualitätssicherungskonzept

Dokumentationskonzept - Testkonzept - Organisatorische Festlegung

Henrik Hillebrand
14.01.2014

Inhaltsverzeichnis

Dokumentationskonzept

Gemäß der Projektvorgabe übernehmen wir die *OntoWiki Coding Standards*, welche auf den Formatvorgaben des *Zend Frameworks* aufbauen.

Allgemeine Formatvorgaben

Folgende allgemeine Vorgaben gelten für alle Textdateien:

- Einrückungen geschehen generell durch vier Leerzeichen anstatt eines Tabulators.
- Zeilen sollten eine Länge von 100 Zeichen im Allgemeinen nicht überschreiten, können aber grundsätzlich bis zu 120 Zeichen enthalten, wenn es dem Entwickler sinnvoller erscheint als die Zeile umzubrechen.
- Dateien, die ausschließlich PHP-Code enthalten sind **nicht** mit einem abschließenden Schluss-Tag („`?>`“) zu versehen.

Namen

Für alle Namen gilt, dass sie keine Umlaute oder andere Sonderzeichen, insbesondere abseits der Standard-ASCII-Tabelle enthalten dürfen.
Soweit möglich sollen Namen so gewählt werden, dass sie die jeweilige Klasse, Funktion oder Variable gut beschreiben. Dieses Ziel wird höher priorisiert als die Vergabe möglichst kurzer Namen.

Dateien

Dateinamen dürfen ausschließlich Buchstaben, Zahlen und Bindestriche („-“) enthalten. Von Zahlen wird aber grundsätzlich abgeraten.
Dateinamen beginnen stets mit einem Großbuchstaben. Wenn sie mehr als ein Wort enthalten sind sie im *CamelCase* fortzuführen. Dies gilt auch für Abkürzungen, also z.B. `Xodx` statt `XODX` oder `xodx`.

Klassen

Jede Klasse ist in einer eigenen Datei zu beschreiben. Klassennamen entsprechen direkt ihrem Dateipfad, wobei ein Unterstrich („_“) anstelle des Verzeichnisseparators verwendet wird.

Funktionen

Funktionsnamen bestehen ausschließlich aus Buchstaben und Zahlen (welche jedoch grundsätzlich zu vermeiden sind). Im Gegensatz zu Klassennamen enthalten Funktionsnamen keine Unterstriche und beginnen stets mit einem Kleinbuchstaben. Bei aus mehreren Wörtern bestehenden Funktionsnamen wird auch hier im CamelCase fortgefahren.

Variablen

Variablen werden im Wesentlichen nach dem gleichen Schema benannt wie Funktionen (siehe oben). Variablen mit der Sichtbarkeit `private` oder `protected` beginnen stets mit einem Unterstrich. Sehr kurze Variablennamen wie `$i` sind nur als Zähler in kurzen Schleifen zu verwenden. Grundsätzlich ist ein aussagekräftiger Name zu bevorzugen.

Konstanten

Konstanten bestehen ausschließlich aus Großbuchstaben und Zahlen, letztere sind wenn möglich zu vermeiden. Aus mehreren Wörtern bestehende Konstanten werden mit Unterstrichen getrennt.

Null

Der `null`-Wert wird generell klein geschrieben.

Klammern

Eine öffnende geschweifte Klammer steht in einer eigenen Zeile, sofern sie eine Klassen- oder Funktionsdeklaration einleitet und in der gleichen Zeile wie die zugehörige Kontrollstruktur, sofern

auf sie ein Anweisungsblock folgt. Schließende geschweifte Klammern stehen stets in einer eigenen Zeile.
Öffnende runde oder eckige Klammern werden stets ohne ein Leerzeichen direkt hinter den Anweisungs- oder Variablennamen gesetzt. Wenn der umklammerte Text umgebrochen wird ist dieser in einer neuen Zeile eingerückt zu beginnen und die schließende Klammer in eine eigene Zeile zu setzen.
Rückgabewerte werden generell nicht eingeklammert.

Strings

Strings, die keine weiteren Variablen enthalten sind in einzelne Hochkommata („'“) zu setzen, sofern der String nicht selbst ebensolche Hochkommata enthält. Alle anderen Strings sind in doppelte Hochkommata („"“) zu setzen.

Kommentare

Um jedem Entwickler einen schnellen Überblick zu verschaffen, ohne den eigentlichen Quellcode lesen oder verstehen zu müssen, ist dieser grundsätzlich zu dokumentieren. Um diese Dokumentation zu extrahieren verwenden wir phpDocumentor – jeder Dokumentationsblock muss also damit kompatibel sein. Weiterhin sind nicht offensichtliche Abläufe direkt im Code zu kommentieren, wobei diese nicht vom phpDocumentor erfasst werden.

Dokumentationsblöcke

Jede Datei, Klasse und Funktion ist mit einem Dokumentationsblock zu versehen, der diese in Prosa beschreibt. Die Dokumentation findet dabei generell in englischer Sprache statt. Die folgenden Beispiele demonstrieren die Mindestanforderungen an einen Dokumentationsblock, weitere Tags sind gemäß der Umstände oder des Urteils des Dokumentierenden hinzuzufügen.

Dateien

Für Dateien gilt das folgende Muster für einen Datei-Dokumentationsblock:

```
/**
 * This file is part of the {@link http://pcai042.informatik.uni-
leipzig.de/~swp14-xodx/ xodx} project.
 *
 * @copyright Copyright (c) 2014, {@link http://aksw.org AKSW}
 * @license http://opensource.org/licenses/gpl-license.php GNU General
Public License (GPL)
 */
```

Dieser Block ist direkt am Anfang einer Datei einzufügen.

Klassen

Für Klassen gilt analog folgendes Muster:

```
/**
 * Short description for class
 *
 * Long description for class (if any) ...
 *
 * @copyright Copyright (c) 2014, {@link http://aksw.org AKSW}
 * @license http://opensource.org/licenses/gpl-license.php GNU General
Public License (GPL)
 * @category Xodx
 * @package Xodx_package
 * @author Carl Committer <carl.committer@gmail.com>
 */
```

Ein solcher Block steht stets direkt vor der Beschreibung einer Klasse.

Funktionen

Neben einer allgemeinen Beschreibung müssen folgende Eigenschaften gesondert dokumentiert werden:

- Sämtliche übergebenen Parameter (`@param`)
- Mögliche Rückgabewerte der Funktion (`@return`)
- Exceptions, die von dieser Funktion geworfen werden können (`@throws`)

Sollten für einen Über- oder Rückgabewert mehrere Datentypen infrage kommen so sind diese durch einen Senkrechtstrich („|“) getrennt aufzulisten.

Kommentare

Einzeilige Kommentare im Quelltext sind mit „`//`“ einzuleiten und nicht zu schließen.
Mehrzeilige Kommentare sind mit „`/* `“ einzuleiten und mit „ `*/`“ zu schließen. Nach der ersten Zeile ist jede weitere Zeile mit „ `* `“ einzuleiten. Das Kommentarende steht dabei in einer eigenen Zeile. Beispiel:

```
/* For one reason or another the code surrounding this comment requires
 * more additional information than a single line could hold.
 */
```

Externe Dokumentation

Es ist unser explizites Ziel eine Oberfläche zu schaffen, die für den gemeinen Benutzer möglichst selbsterklärend ist. Dennoch wird eine Hilfeseite erstellt, die Funktionen erläutert und häufige Fragen („FAQ“) beantwortet. Der User wird hierbei mithilfe von Screenshots oder Screencasts Klick-weise beispielsweise durch den Prozess geleitet eine Gruppe zu erstellen oder Gruppenmitglieder hinzuzufügen.

Testkonzept

Bei der Entwicklung komplexer Software ist es wichtig diese bei der Entwicklung regelmäßig zu testen. Die meisten Softwareprodukte werden in Teams programmiert in denen jedes Mitglied ein Teilprojekt übernimmt. Um Probleme beim Zusammenfügen der Softwarekomponenten zu vermeiden muss jedes Teilprojekt bereits einzeln getestet werden. Dies schließt jedoch mögliche Fehler im Gesamtprodukt nicht aus denn dazu muss auch das gesamte Projekt überprüft werden. Um eine Software zu testen stellen viele Entwicklungsumgebungen bereits vorgefertigte und automatische Tests und Testumgebungen zur Verfügung. Oft ist es auch der Fall, dass eine bereits vorhandene Software weiterentwickelt werden soll, wobei es sinnvoll ist, die dort verwendeten Testframeworks zu nutzen. Doch genauso wichtig wie die automatischen Tests sind manuelle Tests, die auf das Programm abgestimmt sind. Ziel solcher Tests ist es, Fehler im Programmcode zu finden und Schwachstellen bezüglich Kontinuität, Sicherheit etc. zu beseitigen.
Im Allgemeinen sind vier Testabschnitte zu berücksichtigen:

- Komponententests
- Integrationstests
- Systemtests
- Abnahmetests

Komponententests

Wie oben bereits erwähnt ist es wichtig jede einzelne Methode zu testen und mögliche Fehler unabhängig von dem Gesamtprojekt zu eliminieren. Für solche Tests stehen Frameworks wie z.B. JUnit (für Java) oder PHPUnit (für PHP) oder Tools wie Syntaxprüfer zur Verfügung. Solche Tests sollen aber nicht nur Fehler finden, damit man Lösungen für diese erarbeiten kann, sondern auch das Dokumentieren dieser Fehler, um bei ähnlichen Fehlern die Lösung zu erleichtern.

Wir verwenden PHPUnit für diese Komponententests. Jedes Teammitglied, das Klassen implementiert oder vorhandene Klassen erweitert, schreibt entsprechende PHPUnit-Testklassen, die automatisch auf das gewünschte Verhalten testen.

Integrationstests

Nachdem alle Komponenten einzeln erfolgreich auf ihre Funktionalität getestet wurden werden diese zusammengefügt und auf ihre Integrationsfähigkeit getestet. Ziel dieser Tests ist es das Zusammenspiel einzelner Bauteile zu garantieren. Von Vorteil ist es, wenn man hier einzelne Methoden nach und nach zusammenfügt und testet, um so schnell Fehler und ihre Ursachen zu beheben.
Wir führen die Integrationstests durch, indem wir nach dem Zusammenführung im Git-Repository, die vorhandenen Änderungen in unserem Testsystem aktualisieren. Anschließend testen wir die Zusammenführung ausgiebig.

Systemtests

Der Systemtest unterscheidet sich von den bisherigen Tests, denn ab hier wird nicht mehr aus Sicht des Entwicklerteams sondern aus der des Kunden/Anwenders auf die Funktionalität geprüft.
Wichtig dabei ist es auch das fertige Produkt in einer Anwendungsumgebung zu testen in der das Produkt eingesetzt wird. In dieser Phase spielt auch das Lastenheft eine bedeutende Rolle, denn hier zeigt sich, ob das fertige Programm den gegebenen Anforderungen entspricht.
Wir testen bei den Systemtests hier einerseits ähnlich wie bei den Integrationstests, indem wir in unseren Testinstanzen alle aktuellen Änderungen manuell durchtesten. Weiterhin verwenden wir als Testtool „Selenium“, was ein Werkzeug darstellt, um wiederkehrende Abläufe aufzunehmen und diese wiederholt abzuspielen. Somit ersparen wir uns bei den Standard-Aufgaben viel Arbeit, da diese somit vollautomatisch getestet werden können. Außerdem ermöglicht uns Selenium die Durchführung gewisser „Stresstests“, da man dort beliebig viele Instanzen der Abläufe erstellen kann.

Abnahmetest

Der Abnahmetest erfolgt zusammen mit dem Kunden. Hier wird das fertige Produkt ebenfalls in einer Umgebung getestet, die der Anwendungsumgebung entspricht und zusammen mit dem Kunden auf die vorgegebenen Anforderungen überprüft. Sofern hier keine Probleme auftreten kann das Produkt dem Kunden übergeben werden.

Organisatorische Festlegungen

Wöchentlich werden Gruppenmeetings abgehalten, in denen die bisherigen Fortschritte vorgestellt und besprochen, neue Aufgaben verteilt und organisatorische Anliegen abgesprochen werden. Zu diesen Treffen erscheint, wenn möglich, jedes Mitglied der Projektgruppe. Bei krankheitsbedingtem Ausfall oder terminlicher Verhinderung wird dies im Projektforum frühzeitig angekündigt, um mögliche Ausweichtermine in Erwägung zu ziehen. Zusätzlich findet jeden Montag ein PHP-Crashkurs statt, der auf freiwilliger Basis beruht.
Sonstige Absprachen außerhalb der Meetings und organisatorische Planung eben jener findet ebenfalls über das projekteigene Forum oder einen installierten XMPP-Chat statt. Über ersteres werden auch Dokumentenvorlagen, How-To's und andere Hilfsmittel der gesamten Gruppe jederzeit zugänglich, zur Verfügung gestellt.
Zur Organisation und Verteilung des Quellcodes wurde ein GIT-Repository angelegt, über welches jede Code-Änderung eingepflegt wird, und eine Dokumentation erstellt, welche die Einarbeitung in Neuerungen möglichst Effizient gestaltet.
Wie oben angemerkt wird sich zur organisatorischen Vereinfachung an den OntoWiki-Coding-Standard gehalten, welches die Übersicht im Aktualisierungs-Feed des GIT-Repository von unnötigen Umformatierungsänderungen befreit. Für das Einhalten dieses Standards ist jeder Programmierer

selbst verantwortlich. In regelmäßigen Abständen wird eine zusätzliche Prüfung durch den Verantwortlichen der Implementierung gemacht.

Quellen

Webseiten

- https://github.com/AKSW/OntoWiki/wiki/Coding-Standards (16.01.2014)
- http://de.wikipedia.org/wiki/Softwaretest (16.01.2014)